Anexo

Índice

A. *Software* y programación

Este trabajo se ha desarrollado fundamentalmente en `Python` y, en particular, a través de un *notebook* de `Jupyter` del cual se mostrarán algunos fragmentos de código de algunas celdas relevantes. Se ha utilizado `Python` por su versatilidad y popularidad en proyectos de ciencia de datos. Esto hace que existan numerosas librerías especialmente útiles en el análisis de datos y la aplicación de técnicas de *machine learning*. Entre las librerías utilizadas, destacan las siguientes:

- `NumPy`: Proporciona un tratamiento más eficiente y versátil de vectores, matrices y *arrays* multidimensionales más allá de las listas de `Python` por defecto. Además, contiene numerosas funciones de todo tipo compatibles con el formato de *array* que introduce la librería y con aplicación en la manipulación de todo tipo de datos y su tratamiento mediante álgebra lineal.

  Disponible en `https://pypi.org/project/numpy`

- `Pandas`: Librería fundamental en el área de la ciencia de datos que ofrece estructuras de datos y operaciones que facilitan enormemente el procesamiento, la búsqueda y el filtrado de datos. Para ello, introduce el tipo de datos *DataFrame* que facilita la agrupación de datos en tablas indexadas sobre las que aplicar todo tipo de funciones y transformaciones.

  Disponible en `https://pypi.org/project/pandas`

- `matplotlib`: Librería de generación de gráficas a partir de *arrays* con numerosas funciones de graficación de histogramas, diagramas de dispersión y representaciones bidimensionales y tridimensionales de todo tipo con una amplia variedad de parámetros para elegir el aspecto de las gráficas.

  Disponible en `https://pypi.org/project/matplotlib`

- `traj-dist`: Como complemento al artículo `https://arxiv.org/pdf/1508.04904.pdf`, sirve para evaluar varios algoritmos populares que miden distancias entre pares de trayectorias caracterizadas por *arrays* de no necesariamente la misma longitud. Incluye las distancias SSPD, OWD, Hausdorff, Fréchet, Fréchet discreto, DTW, LCSS, ERP y EDR.

  Disponible en `https://pypi.org/project/traj-dist`

- `seaborn`: Otra librería de visualización basada en `matplotlib` que proporciona gráficos muy ilustrativos de información estadística. Por ejemplo, en el trabajo se ha utilizado para obetener matrices de gráficas y matrices de confusión con formato de mapas de calor.

  Disponible en `https://pypi.org/project/seaborn`

- `scikit-learn`: Una de las librerías más populares en el mundo del *machine learning* que proporciona algunas funciones relacionadas con el diseño de modelos de árboles de decisión, redes neuronales y otras herramientas relacionadas con la selección de modelos, preprocesamiento de datos, medición de la precisión y los errores, validación de modelos, etc.

  Disponible en `https://pypi.org/project/scikit-learn`

- `TensorFlow`: Otra librería que suministra funciones de inteligencia artificial. Cuenta con más funciones adaptadas al mundo del *deep learning*. Introduce un formato particular de datos conocido como *tensor*.

  Disponible en `https://pypi.org/project/tensorflow`

- `Keras`: Librería de redes neuronales capaz de ejecutarse sobre `TensorFlow` con un diseño más amigable con el usuario.

  Disponible en `https://pypi.org/project/keras`

- `Yellowbrick`: Librería especializada en las representaciones gráficas de elementos de *machine learning* y con una implementación especialmente adecuada de las curvas ROC multiclase.

  Disponible en `https://pypi.org/project/yellowbrick`

- `XGBoost`: Proporciona algoritmos basados en los métodos de *gradient boosting*. En el caso de algoritmos basados en árboles de decisión, son relativamente populares por poder llegar a superar a los árboles clásicos y a los bosques en algunos casos por lo que merecía la pena experimentar con su implementación.

  Disponible en `https://pypi.org/project/xgboost`

Se muestran a continuación algunas funciones importadas de módulos de las librerías mencionadas para su aplicación a lo largo de todo el trabajo:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import traj_dist.distance as tdist
import random
import seaborn as sn
from sklearn import tree
from sklearn.datasets import load_iris, make_regression
from sklearn.cluster import KMeans
from sklearn.tree import DecisionTreeClassifier, export_graphviz,
    DecisionTreeRegressor
from sklearn.model_selection import train_test_split, validation_curve, KFold,
    GridSearchCV, cross_validate
from sklearn.metrics import confusion_matrix, accuracy_score,
    mean_absolute_error
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,
    GradientBoostingClassifier
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.utils import resample
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from yellowbrick.classifier import ROCAUC
from xgboost import XGBClassifier, XGBRegressor
```

Asimismo, la visualización y tratamiento inicial de los datos geográficamente referenciados se ha hecho a través del Sistema de Información Geográfica `QGIS`. Los datos disponibles con los que se ha desarrollado este trabajo se encuentran en formatos `LineString` y `MultiLineString` adecuados para la carga directa de los datos en `QGIS` y así procesar simultáneamente cientos de datos de trayectorias geográficas, la localización de cada tipo de contenedor y otra información asociada geográficamente a cada orden de trabajo. No obstante, una parte importante de este trabajo ha sido diseñar una integración adecuada entre `Python` y `QGIS` para un tratamiento eficiente de los datos dependiendo de los momentos en los que puede ser necesaria una representación gráfica de la información geográfica y los momentos en los que se requiere un tratamiento estrictamente numérico y estadístico de los datos.

`QGIS` está disponible en `https://www.qgis.org/es/site/forusers/download.html`

B. Acerca del preprocesamiento de datos

B.1. Filtrado de datos

Para preparar los datos, se debe automatizar el proceso de lectura de datos, formateo, filtrado de las órdenes para las que se dispone información completa (real, teórica y contenedores) y descarte de información duplicada:

```
def filtra_ordenes(csvteoricas,csvreales,csvcoordenadas):
    dfteoricas=pd.read_csv(csvteoricas,sep=";")
    dfreales=pd.read_csv(csvreales,sep=";")
    dfcoordenadas=pd.read_csv(csvcoordenadas,sep=";")

    dfcoordenadas["Latitude"] = dfcoordenadas["Latitude"].astype(str).str.
    replace(",",".")
    dfcoordenadas["Longitude"] = dfcoordenadas["Longitude"].astype(str).str.
    replace(",",".")

    todasreales = pd.Series(dfreales['WorkOrderId'])
    duplicadas = list(todasreales[todasreales.duplicated()])

    ordenesteoricas=np.unique(dfteoricas['WorkOrderId'])
    ordenesreales = np.array([x for x in dfreales['WorkOrderId'] if x not in
    duplicadas])
    ordenescoordenadas=np.unique(dfcoordenadas['WorkOrderId'])
    ordenes=[ordenesteoricas,ordenesreales,ordenescoordenadas]
    ordenescomunes=np.sort(list(set.intersection(*map(set,ordenes))))

    filtroteorico=dfteoricas[dfteoricas['WorkOrderId'].isin(ordenescomunes)]
    filtroreal=dfreales[dfreales['WorkOrderId'].isin(ordenescomunes)]
    filtrocoord=dfcoordenadas[dfcoordenadas['WorkOrderId'].isin(ordenescomunes)]

    filtrocoordfinal = pd.DataFrame(columns=filtrocoord.columns)

    for orden in ordenescomunes:
```

```
        filtrocoordfinal = filtrocoordfinal.append(filtrocoord[filtrocoord['
    WorkOrderId']==orden].drop_duplicates(subset=['ContainerId']),ignore_index=
    True)

    filtroteorico.to_csv('FiltroTeorico_'+csvteoricas,sep=';',index=False)
    filtroreal.to_csv('FiltroReal_'+csvreales,sep=';',index=False)
    filtrocoordfinal.to_csv('FiltroCoord_'+csvcoordenadas,sep=';',index=False)
```

B.2. Generación de variables

A continuación, se muestra la función principal utilizada para crear las variables a partir de la lectura de 3 ficheros `csv` de rutas teóricas, rutas reales y coordenadas de contenedores, la extracción de su información y todas las transformaciones comentadas en el trabajo junto con el *clustering*.

```
def clasifica1(cen_x):
    if cen_x==centroides[0]:
        return 0
    if cen_x==centroides[1]:
        return 1
    if cen_x==centroides[2]:
        return 2
    if cen_x==centroides[3]:
        return 3

def clasifica2(Bondad):
    if Bondad>=0 and Bondad<min1:
        return 0
    if Bondad>=min1 and Bondad<min2:
        return 1
    if Bondad>=min2 and Bondad<min3:
        return 2
    if Bondad>=min3:
        return 3

def calcula_variables(csvteoricas,csvreales,csvcoordenadas):

    ordenes = pd.read_csv(csvcoordenadas,sep=';')

    ordenesag = ordenes.groupby('WorkOrderId').agg({
        'Scheduled': 'sum',
        'Collected': 'sum'
    })
    ordenesag['NContenedores']=ordenes.groupby('WorkOrderId').count().iloc[:,2]
    ordenesag['01']=ordenesag['NContenedores']-ordenesag['Scheduled']
    ordenesag['10']=ordenesag['NContenedores']-ordenesag['Collected']
    ordenesag['11']=ordenesag['NContenedores']-ordenesag['01']-ordenesag['10']
    ordenesag=ordenesag.drop(columns=['Scheduled','Collected'])
    ordenesag['Ruido'] = None
    ordenesag['BienHechos']=ordenesag['11']/(ordenesag['11']+ordenesag['10'])
    ordenesag['Similitud'] = None
```

```
37      ordenesag['Adicionales']=ordenesag['01']/(ordenesag['11']+ordenesag['10']+
    ordenesag['01'])
38      ordenesag['RatioLongitudes'] = None
39
40      teoricas = pd.read_csv(csvteoricas,sep=";")
41      reales = pd.read_csv(csvreales,sep=";")
42
43      for j in range(len(reales['WorkOrderId'])):
44          wktr = reales['WKT'].iloc[j].replace("LINESTRING(" , "").replace(")","")
    .replace(","," ").split()
45          wktreales = []
46          for i in range(0,len(wktr),2):
47              wktreales.append([float(wktr[i]),float(wktr[i+1])])
48          longitudes_r = np.sqrt(np.sum(np.diff(np.array(wktreales), axis=0)**2,
    axis=1))
49          longitud_real = np.sum(longitudes_r)
50
51          wktt = teoricas['WKT'].iloc[j].replace("MULTILINESTRING((" , "").replace
    ("))","").replace(","," ").split()
52          wktteoricas = []
53          for i in range(0,len(wktt),2):
54              wktteoricas.append([float(wktt[i]),float(wktt[i+1])])
55          longitudes_t = np.sqrt(np.sum(np.diff(np.array(wktteoricas), axis=0)**2,
     axis=1))
56          longitud_teorica = np.sum(longitudes_t)
57
58          ratio = np.abs(longitud_real-longitud_teorica)/longitud_teorica
59          ordenesag['RatioLongitudes'].iloc[j]=ratio
60
61          distancia = tdist.sspd(np.array(wktreales),np.array(wktteoricas),'
    euclidean')
62          ordenesag['Similitud'].iloc[j]=distancia
63
64          ordenesag['Ruido'].iloc[j]=np.random.normal(0,0.7)
65
66      ordenesag['Similitud']=0.001/(0.001+ordenesag['Similitud'])
67
68      ordenesag['RatioLongitudes']=(ordenesag['RatioLongitudes']-ordenesag['
    RatioLongitudes'].min())/(ordenesag['RatioLongitudes'].max()-ordenesag['
    RatioLongitudes'].min())
69
70      ordenesag['Bondad']=10*ordenesag['BienHechos']+4*ordenesag['Similitud']+2*
    ordenesag['Adicionales']-1*ordenesag['RatioLongitudes']+ordenesag['Ruido']
71
72      ordenesag['Bondad']=10*(ordenesag['Bondad']-ordenesag['Bondad'].min())/(
    ordenesag['Bondad'].max()-ordenesag['Bondad'].min())
73
74      mezcla = ordenesag.copy()
75
76      kmeans = KMeans(n_clusters=4, random_state=0)
77      mezcla['cluster'] = kmeans.fit_predict(mezcla[['Bondad']])
78      centroids = kmeans.cluster_centers_
79      cen_x = [i[0] for i in centroids]
80      mezcla['cen_x'] = mezcla.cluster.map({0:cen_x[0], 1:cen_x[1], 2:cen_x[2], 3:
    cen_x[3]})
```

```
81
82      global centroides
83
84      centroides = np.sort(np.unique(mezcla['cen_x']))
85      colors = ['#DF2020', '#81DF20', '#2095DF', '#20DF95']
86      mezcla['c'] = mezcla.cluster.map({0:colors[0], 1:colors[1], 2:colors[2], 3:
        colors[3]})
87      plt.scatter(mezcla.Bondad,mezcla.Bondad, c=mezcla.c, alpha = 0.6, s=10)
88      ordenesag['Resultado']=mezcla['cen_x'].apply(clasifica1)
89
90      global min0
91      global min1
92      global min2
93      global min3
94
95      min0 = min(ordenesag[ordenesag['Resultado']==0]['Bondad'])
96      min1 = min(ordenesag[ordenesag['Resultado']==1]['Bondad'])
97      min2 = min(ordenesag[ordenesag['Resultado']==2]['Bondad'])
98      min3 = min(ordenesag[ordenesag['Resultado']==3]['Bondad'])
99
100     print(min0)
101     print(min1)
102     print(min2)
103     print(min3)
104
105     ordenesag['Bondad']=ordenesag['Bondad'].astype(float).round(0)
106
107     ordenesag['Resultado']=ordenesag['Bondad'].apply(clasifica2)
108
109     return ordenesag
```

C. Acerca de la similitud de trayectorias

C.1. Densificación de trayectorias

En el transcurso de este trabajo, se ha comprobado que la SSPD es el algoritmo óptimo en el cálculo del grado de similitud de los pares de trayectorias disponibles en este conjunto de datos. Se puede mejorar aún más la precisión de las distancias si se densifica con más puntos cada una de las trayectorias. Estas trayectorias están formadas por una serie de puntos unidos por rectas, pero si queremos rellenar dichas rectas con más puntos equiespaciados para reconstruir más densamente las trayectorias, puede emplearse la siguiente función:

```
1 def densificar(coordenadas, segmentos):
2     denso=coordenadas
3     for i in range(len(coordenadas)-1):
4         denso=np.insert(denso,i*segmentos+1,np.array([list(a) for a in zip(np.
      linspace(coordenadas[i][0],coordenadas[i+1][0],segmentos,endpoint=False)
      [1:],np.linspace(coordenadas[i][1],coordenadas[i+1][1],segmentos,endpoint=
      False)[1:])]),0)
```

```
5     return denso
```

donde `coordenadas` es el *array* de puntos que conforman una trayectoria y `segmentos` es el número de segmentos que queremos que haya entre dos puntos consecutivos del *array* original (es decir, que haya `segmentos`-1 puntos nuevos entre cada par de puntos consecutivos originales).

C.2. Medidas alternativas del grado de similitud

Hasta comprobar la idoneidad de la SSPD, se analizaron numerosos algoritmos populares que funcionarían bien en otros problemas específicos. Podemos clasificar estos algoritmos en dos tipos:

- *Warping-based*: Son aquellas distancias que tienen en cuenta el indexado temporal de cada punto. Por ejemplo, las distancias DTW, LCSS, EDR y ERP.
- *Shape-based*: Son aquellas distancias que únicamente tienen en cuenta la forma geométrica de las trayectorias. Por ejemplo, las distancias Hausdorff, Fréchet y SSPD.

Empecemos caracterizando los cuatro populares algoritmos *warping-based* mencionados:

- ***Dynamic Time Warping* (DTW)**
- ***Longest Common SubSequence* (LCSS)**
- ***Edit Distance on Real sequence* (EDR)**
- ***Edit distance with Real Penalty* (ERP)**

En las descripciones matemáticas que siguen, se quiere calcular la distancia entre dos trayectorias T^i y T^j donde n^i es el número de puntos de T^i y n^j es el número de puntos de T^j. El punto k-ésimo de la trayectoria T^i es p_k^i y el punto k-ésimo de la trayectoria T^j es p_k^j. Asimismo, $rest(T^i)$ (respectivamente, $rest(T^j)$) es la trayectoria T^i (respectivamente, T^j), pero sin su primer punto. Además, LCSS y EDR requieren la especificación de un umbral espacial ε_d y ERP toma un parámetro g como valor de referencia para penalizar *gaps* (se dan cuando hay puntos que no se emparejan con ningún otro punto).

La definición de los 4 aparece resumida en la siguiente tabla:

NAME	Cost function $\delta_{NAME}(p_1, p_2)$ =	Distance $NAME(T^i, T^j)$ =
DTW	$\|p_1p_2\|_2$	$= \begin{cases} 0 & \text{if } n^i = n^j = 0 \\ \infty & \text{if } n^i = 0 \text{ or } n^j = 0 \\ \delta_{DTW}(p_1^i, p_1^j) + \min \left\{ \begin{array}{l} DTW(rest(T^i), rest(T^j)), \\ DTW(rest(T^i), T^j)), \\ DTW(T^i, rest(T^j) \end{array} \right\} & \text{otherwise} \end{cases}$
LCSS	$\begin{cases} 1 & \text{if } \|p_1p_2\|_2 < \varepsilon_d \\ 0 & \text{if } p_1 \text{ or } p_2 \text{ is a gap} \\ 0 & \text{otherwise} \end{cases}$	$= \begin{cases} 0 & \text{if } n^i = 0 \text{ or } n^j = 0 \\ LCSS(rest(T^i), rest(T^j)) + \delta_{LCSS}(p_1^i, p_1^j) & \text{if } \delta_{LCSS}(p_1^i, p_1^j) = 1 \\ \max \left\{ \begin{array}{lll} LCSS(rest(T^i), T^j)) & + & \delta_{LCSS}(p_1^i, gap), \\ LCSS(T^i, rest(T^j)) & + & \delta_{LCSS}(gap, p_1^j) \end{array} \right\} & \text{otherwise} \end{cases}$
EDR	$\begin{cases} 0 & \text{if } \|p_1p_2\|_2 < \varepsilon_d \\ 1 & \text{if } p_1 \text{ or } p_2 \text{ is a gap} \\ 1 & \text{otherwise} \end{cases}$	$= \begin{cases} n^i & \text{if } n^j = 0 \\ n^j & \text{if } n^i = 0 \\ EDR(rest(T^i), rest(T^j)) & \text{if } \delta_{EDR}(p_1^i, p_1^j) = 0 \\ \min \left\{ \begin{array}{lll} EDR(rest(T^i), rest(T^j)) & + & \delta_{EDR}(p_1^i, p_1^j), \\ EDR(rest(T^i), T^j)) & + & \delta_{EDR}(p_1^i, gap), \\ EDR(T^i, rest(T^j)) & + & \delta_{EDR}(gap, p_1^j) \end{array} \right\} & \text{otherwise} \end{cases}$
ERP	$\begin{cases} \|p_1p_2\|_2 & \text{if } p_1, p_2 \text{ are not gaps} \\ \|p_1g\|_2 & \text{if } p_2 \text{ is a gap} \\ \|gp_2\|_2 & \text{if } p_1 \text{ is a gap} \end{cases}$	$= \begin{cases} \sum_{k=1}^{n^i} \|p_k^i g\|_2 & \text{if } n^j = 0 \\ \sum_{l=1}^{n^j} \|p_l^j g\|_2 & \text{if } n^i = 0 \\ \min \left\{ \begin{array}{lll} ERP(rest(T^i), rest(T^j)) & + & \delta_{ERP}(p_1^i, p_1^j), \\ ERP(rest(T^i), T^j)) & + & \delta_{ERP}(p_1^i, gap), \\ ERP(T^i, rest(T^j)) & + & \delta_{ERP}(gap, p_1^j) \end{array} \right\} & \text{otherwise} \end{cases}$

Tabla 1: Cuatro distancias *warping-based*: DTW, LCSS, EDR y ERP. A la izquierda, funciones de coste δ que se definen aparte para implementarse en las definiciones de las distancias de la parte derecha.

Para nuestro problema concreto, los algoritmos *warping-based* arrojarían resultados poco fiables debido a que intentarían emparejar los puntos de las dos trayectorias según su ordenación en el tiempo.

Como la variable que deseamos obtener es el grado de similitud entre dos rutas (la real y la teórica) basándonos en su forma, parece lógico el estudio de los algoritmos *shape-based* como los que se describen a continuación:

- **Distancia de Hausdorff**:

$$D_{Haus}(T^1, T^2) = \text{máx} \left\{ \underset{\substack{i_1 \in [1,\ldots,n^1] \\ j_2 \in [1,\ldots,n^2-1]}}{\text{máx}} \left\{ D_{ps}\left(p_{i_1}^1, s_{j_2}^2\right) \right\}, \underset{\substack{j_1 \in [1,\ldots,n^1-1] \\ i_2 \in [1,\ldots,n^2]}}{\text{máx}} \left\{ D_{ps}\left(p_{i_2}^2, s_{j_1}^1\right) \right\} \right\} \quad (1)$$

  donde $D_{ps}\left(p_{i_1}^1, s_{j_2}^2\right)$ es la distancia del punto $p_{i_1}^1$ al segmento $s_{j_2}^2$ y $D_{ps}\left(p_{i_2}^2, s_{j_1}^1\right)$ es la distancia del punto $p_{i_2}^2$ al segmento $s_{j_1}^1$.

- **Distancia de Fréchet**:

  Dadas dos curvas A y B, se define esta distancia como el ínfimo sobre todas las reparametrizaciones α y β de $[0,1]$ del máximo sobre todos los $t \in [0,1]$ de la distancia entre $A(\alpha(t))$ y $B(\beta(t))$:

$$D_{Frec}(A, B) = \underset{\alpha,\beta}{\text{ínf}} \underset{t \in [0,1]}{\text{máx}} \{d\left(A(\alpha(t)), B(\beta(t))\right)\} \quad (2)$$

  Alternativamente, existe una versión discreta de este algoritmo.

- ***Symmetrized Segment-Path Distance*** **(SSPD)**:

  La descrita en el trabajo principal y que resultó ser la que proporcionaba estimaciones más precisas y con motivos teóricos para creer en su eficacia para nuestro problema concreto.

  Como en nuestro caso buscamos variables normalizadas para evitar problemas de sensibilidad de escala en árboles de decisión y redes neuronales, tan sólo será necesario fijar un cierto parámetro que garantice una adecuada transformación. Esto se describe en la siguiente subsección.

C.3. Determinación del parámetro de normalización de SSPD

Es fundamental convertir el valor de todas las variables de entrada a un intervalo acotado para evitar el problema de la sensibilidad de escala que presentan algunos modelos de *machine learning*. Si la distancia SSPD es un número entre cero e infinito, se determinó que la transformación más acertada dadas las características del problema era

$$\frac{\xi}{\xi + SSPD} \tag{3}$$

Esta transformación es monótona decreciente, por lo que invierte la ordenación de las distancias originales sin mezclarlas y asignará un grado de similitud alto a una distancia baja, y viceversa. Esto último ocurrirá tan solo si se elige adecuadamente el parámetro ξ porque su mala elección en orden de magnitud hace que los valores del grado de similitud se agrupen en un extremo del intervalo $(0, 1)$. La elección del parámetro óptimo podemos hacerla estudiando la abundancia de datos de distancias en cada orden de magnitud. Empleando la distancia euclídea, observamos que en el conjunto de datos hay distancias en los órdenes de magnitud 10^{-5}, 10^{-4}, 10^{-3} y 10^{-2}. En particular, entre las 623 órdenes de trabajo analizadas, se tiene la siguiente distribución:

Orden de magnitud de la distancia	**Cantidad de órdenes de trabajo**
10^{-2}	33
10^{-3}	169
10^{-4}	363
10^{-5}	58

Tabla 2: Número de órdenes de trabajo de acuerdo al orden de magnitud en el que se encuentra su SSPD asociada.

Recordemos que cuanto menor es esta distancia, más similares son dos rutas. Realizamos una inspección cualitativa de cómo de similares son las trayectorias geométricas de órdenes de trabajo que posean distancias en cada uno de los cuatro órdenes de magnitud descritos. Finalmente, llegamos a la conclusión de que estos 4 órdenes de magnitud son plenamente identificables con precisamente las 4 clases que hemos definido para el entrenamiento de los modelos clasificadores. Es decir, una distancia del orden de 10^{-2} sería una distancia ***mala***, una distancia del orden de

10^{-3} sería una distancia ***regular***, una distancia del orden de 10^{-4} sería una distancia ***razonablemente buena*** y una distancia del orden de 10^{-5} sería una distancia ***muy buena***. Por supuesto, como se ha visto en el trabajo principal, esto no es suficiente para realizar la clasificación de las órdenes de trabajo porque influyen otras tres variables con diferentes pesos, pero nos da una idea de que alrededor de 2/3 de la órdenes presentan rutas GPS con una similitud aceptable. Para encontrar el parámetro ξ de la transformación (3) que enviará estas distancias al intervalo (0,1) con una cierta lógica, vamos a suponer que todas las distancias de los órdenes 10^{-2} y 10^{-3} (en total, 202) deben tener asociadas un grado de similitud por debajo de 0,5 y que todas las distancias de los órdenes 10^{-4} y 10^{-5} (en total, 421) deben tener asociadas un grado de similitud por encima de 0,5. El parámetro que permite precisamente esta colocación de los valores es $\xi = 0,001$. Para asegurarnos de que realmente la transformación (3) es realista, hemos de hacer una calibración fijando manualmente unos pocos valores de referencia *a ojo* en ciertas órdenes de trabajo para ver que realmente nuestra transformación es capaz de predecir los grados de similitud que una persona asignaría sin ayuda de un ordenador. Si tomamos un conjunto de 10 órdenes de trabajo con distancias suficientemente variadas, puede comprobarse que el grado de similitud que una persona asignaría manualmente a una pareja de ruta teórica y ruta real es suficientemente parecido al que predice la función (3) con $\xi = 0,001$.

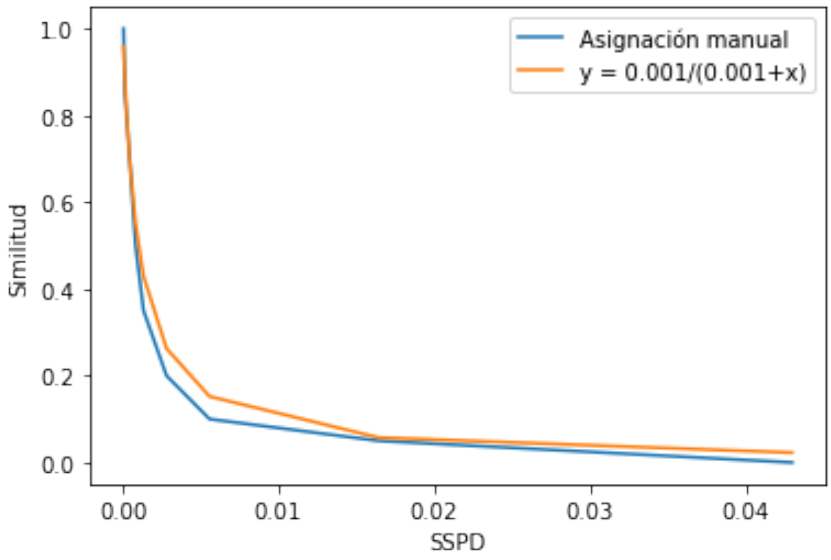

Figura 1: Grado de similitud en función de la distancia SSPD.

Por tanto, hemos conseguido ajustar una magnitud que *a priori* podría parecer totalmente subjetiva o muy difícil de calcular con suficiente precisión y ahora puede ofrecer resultados objetivos.

D. Acerca de los árboles de decisión

A continuación, se muestra una pequeña ampliación de los análisis llevados a cabo durante el trabajo relacionados con los árboles de decisión.

D.1. Profundidad de un árbol

En el trabajo se ha enfatizado que una de las grandes ventajas de los árboles de decisión es su explicabilidad frente a otros modelos con algoritmos más sofisticados que son menos interpretables. Un simple vistazo a un árbol de decisión nos da una idea muy visual de cómo clasificar conjuntos de datos. No obstante, esto sólo es posible cuando el árbol está adecuadamente podado. En el trabajo, se muestra una gráfica de un árbol desarrollado hasta la tercera generación (es decir, tres filas más la del nodo raíz). Esto proporciona una fácil interpretación en función de dos de las variables del modelo y la precisión del modelo puede ser considerada moderadamente buena. Sin embargo, si permitiéramos el desarrollo máximo del árbol hasta que todos los nodos terminales tuvieran una impureza de Gini nula, obtendríamos el siguiente árbol:

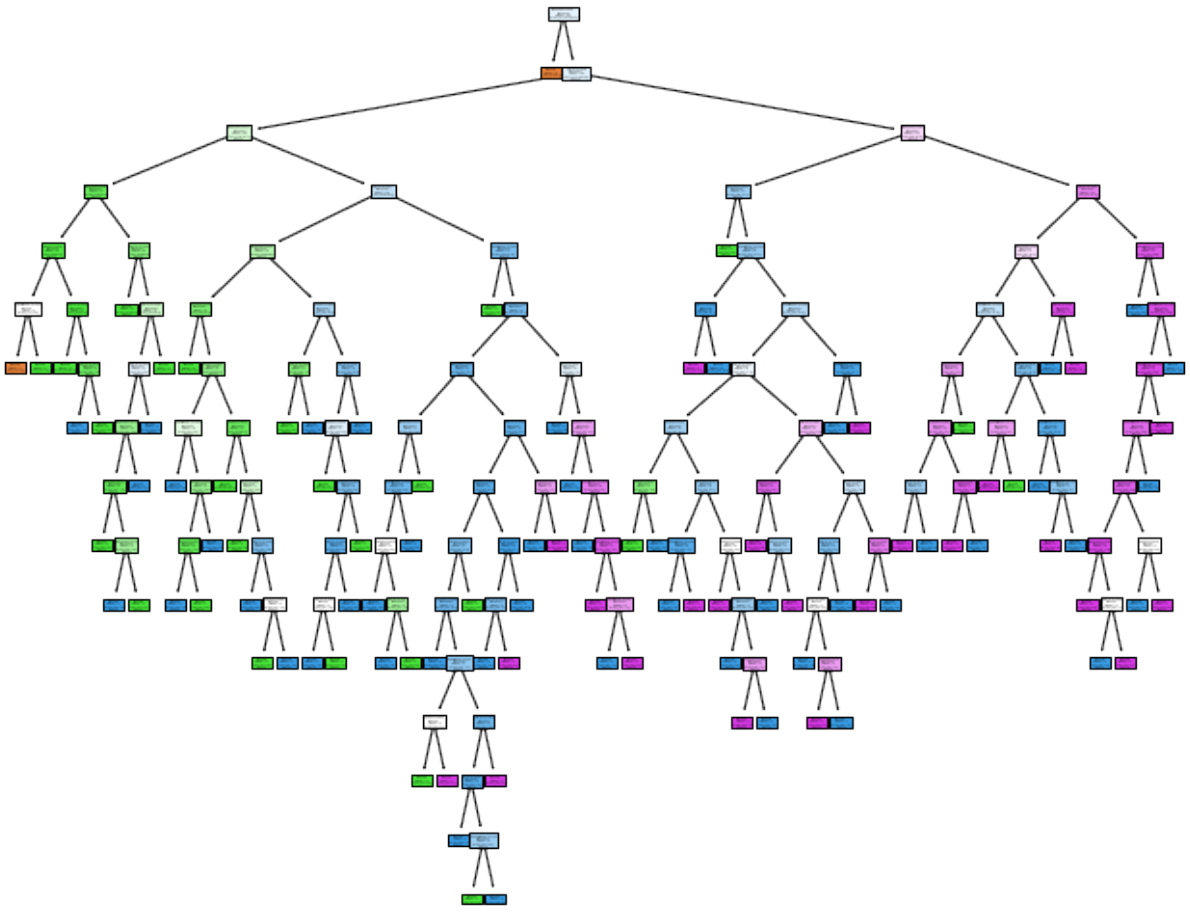

Figura 2: Árbol de decisión si no se hubiera realizado la poda del árbol del trabajo principal.

No se pretende hacer hincapié en la lectura de cada nodo. Lo que sí puede observarse con claridad es que se pierde totalmente la legibilidad. Resultaría muy costoso extraer todas las condiciones que se deducen a través de las proposiciones lógicas asociadas a semejante cantidad de nodos. Además, no necesariamente obtendríamos un modelo con mayor precisión porque un árbol demasiado extenso puede sobreajustar los datos, es decir, aprender demasiado bien los patrones del conjunto de datos de entrenamiento, pero generalizar muy mal y fracasar en las predicciones de cualquier nuevo conjunto de datos. Así, el aprendizaje no tendría ninguna utilidad práctica posteriormente.

D.2. Variabilidad en árboles

No sólo se puede generar un árbol de decisión único. Según el fragmento específico de datos que elijamos para entrenar y para validar, pueden generarse árboles ligeramente distintos. En general, para un conjunto de datos suficientemente grande no deberíamos esperar una variabilidad extremadamente elevada, pero sí puede haber casos aislados que den lugar a resultados inesperados.

Por ejemplo, para el conjunto de datos utilizado, la forma de un árbol de decisión típico y abundante es la que se ha presentado en el trabajo principal dependiente de las variables `BienHechos` y `Similitud`:

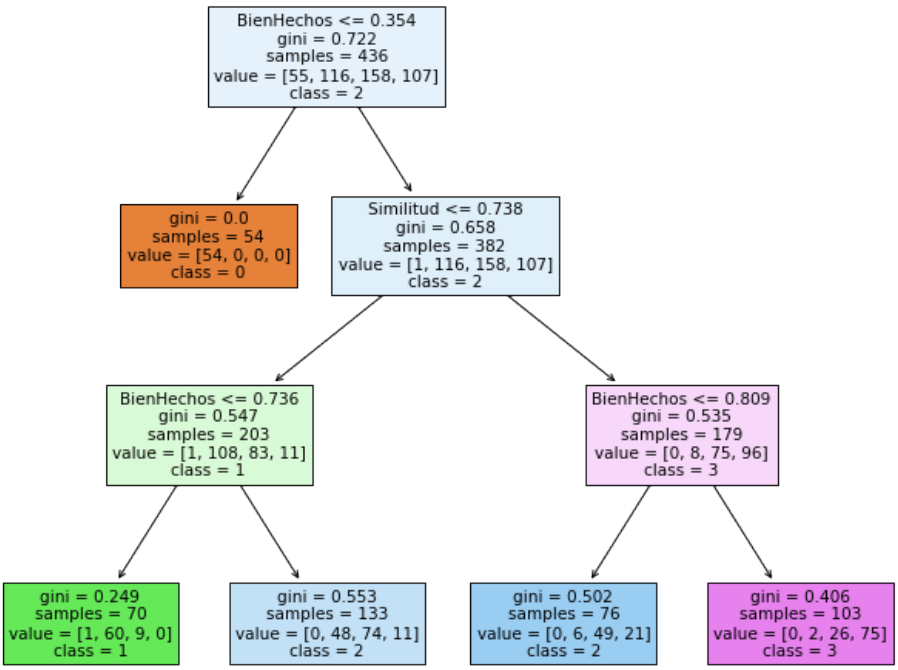

Figura 3: Árbol de decisión principal.

Pero en casos aislados concretos puede llegar a obtenerse algún árbol de la siguiente forma:

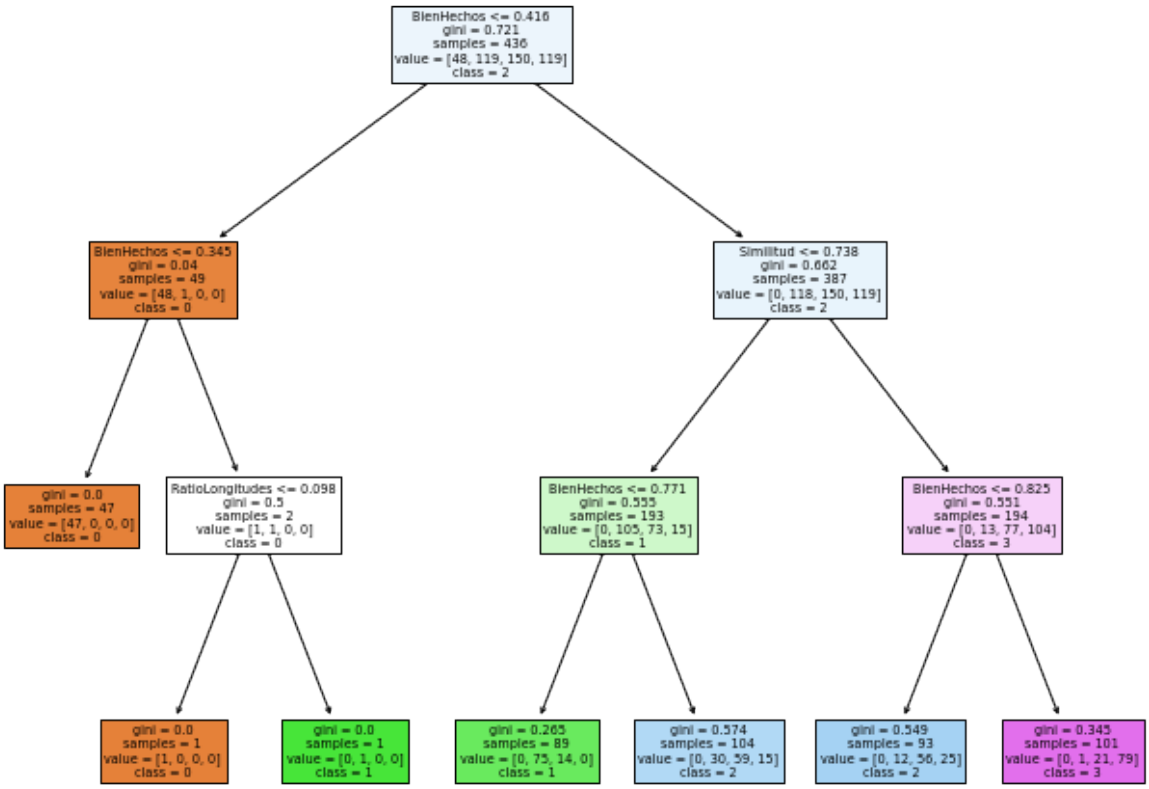

Figura 4: Árbol de decisión alternativo.

donde el algoritmo ha considerado relevante incluir una tercera variable: `RatioLongitudes`. En este caso, vemos que la ha empleado para discriminar entre dos muestras concretas, y ni siquiera de forma correcta porque el modelo debe premiar parejas de rutas teóricas y rutas reales con un bajo valor de `RatioLongitudes`. Este problema puede presentarse en situaciones específicas, así como también pueden darse pequeñas variaciones numéricas en los umbrales de decisión de los nodos según el árbol elegido, incluso si tienen la misma forma. Por ello, el estudio de los árboles de decisión es importante si puede demostrarse que la forma de los árboles es razonablemente estable para un conjunto de datos y si se quiere tener una idea aproximada y visual de cómo sería una clasificación típica. No obstante, la motivación de alcanzar porcentajes de acierto mayores justifica la necesidad de probar modelos algo más elaborados.

D.3. Selección de parámetros del bosque aleatorio

Un bosque aleatorio se forma a partir de las predicciones de múltiples árboles de decisión. No existe una fórmula estándar que permita determinar de antemano el número de árboles óptimo, pero podemos evaluar el rendimiento del modelo según el número de árboles seleccionado:

```
scores = []
for k in range(1, 100):
    rfc = RandomForestClassifier(n_estimators=k,max_depth=3,n_jobs=-1,
    random_state=42)
    rfc.fit(X_train, y_train)
    y_pred = rfc.predict(X_test)
    scores.append(accuracy_score(y_test, y_pred))
```

En nuestro caso, se obtiene el siguiente resultado:

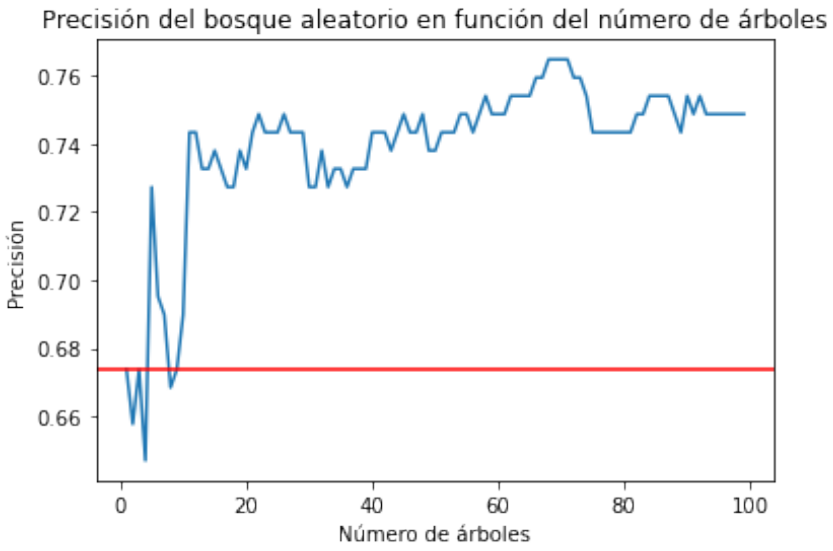

Figura 5: En azul, precisión del bosque aleatorio en función del número de árboles de decisión (estimadores). En rojo, precisión del árbol de decisión individual mostrado en el trabajo.

Vemos que hay altas variaciones en la precisión cuando el número de árboles es bajo. Conforme el número aumenta, la precisión empieza a estabilizarse en un rango de valores más altos que la precisión que conseguíamos con un único árbol de decisión. A la vista de la gráfica anterior, se ha visto que la precisión se maximiza tomando 70 árboles, que es el número elegido en el trabajo. No obstante, tal y como luego se demuestra mediante *bootstrapping*, la capacidad real del modelo es menor que la que se logra con un único bosque aleatorio de 70 árboles. Otra métrica comúnmente utilizada para justificar el número de árboles elegidos y evaluar el rendimiento es el error *out-of-bag* (OOB) que debería reducirse conforme aumenta el número de árboles.

Con un conjunto de datos más grande, se podría acotar más el rango de precisiones en el que oscila el modelo para un número elevado de árboles y podríamos notar, en general, un crecimiento más monótono salvo pequeñas subidas y bajadas debido a la aleatoriedad.

D.4. Superficies de decisión del bosque aleatorio

No es útil representar 70 árboles de decisión individuales, por lo que decimos que un bosque aleatorio es menos interpretable que un árbol de decisión. Sin embargo, podemos tomar las

variables de cualquier modelo por parejas y representar las superficies de decisión. Esto es un tipo de representación que nos permite visualizar cómo un modelo de clasificación divide el espacio de variables de acuerdo a sus predicciones y analizar superficialmente sus puntos fuertes y debilidades. Se trata de entrenar un bosque aleatorio mediante parejas de variables y, para cada pareja, hallar los límites que separan las 4 clases según los umbrales que decide el bosque.

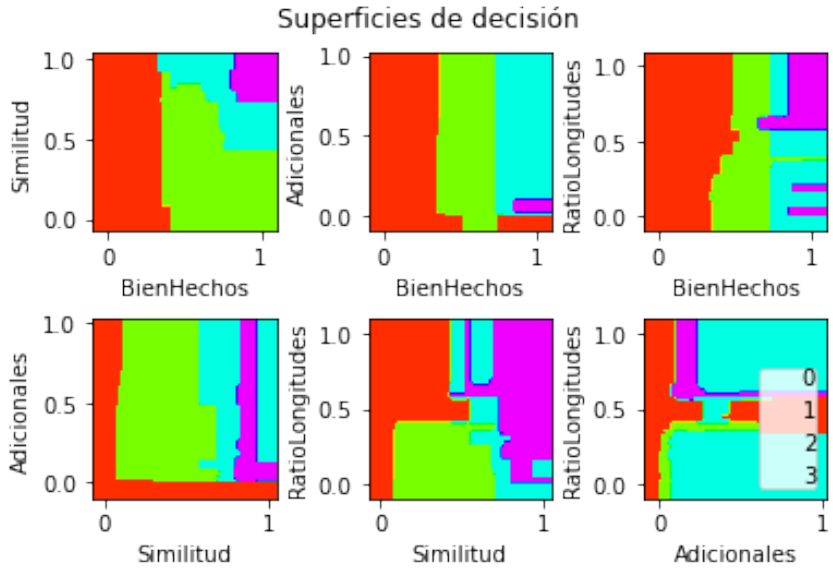

Figura 6: Superficies de decisión inferidas entrenando bosques aleatorios con dos variables. En naranja, la clase 0. En verde claro, la clase 1. En cian, la clase 2. En fucsia, la clase 3.

Tras ello, pintamos encima los datos usados para entrenar y observamos las correspondencias.

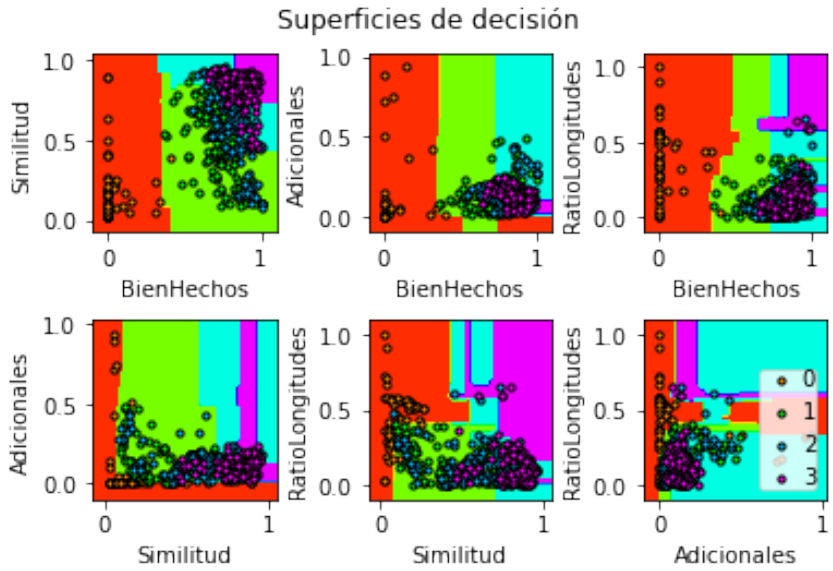

Figura 7: Mismas superficies con los puntos del conjunto de datos con el color adecuado según la clase a la que sabemos que pertenecen.

Las superficies son un intento de distinguir zonas continuas que tienen asociada una misma clase, con mayor o menor grado de éxito. Dada la complejidad del conjunto de datos y que los puntos de una misma clase no están completamente separados de los de otra clase, algunas de las superficies adoptan una forma compleja.

Ante todo, hay que observar que las superficies están dibujadas en cuadrados $[0,1] \times [0,1]$ ya que todas las variables se encuentran en el intervalo $[0,1]$, pero no necesariamente tenemos datos en todas las zonas de un cuadrado. De hecho, en muchos casos vemos que la gran mayoría de los datos se agrupan en zonas muy específicas (esquinas y bordes). El algoritmo hace un intento de extrapolar las superficies a áreas donde no hay puntos, sin que estas predicciones sean necesariamente correctas de acuerdo a nuestra intuición.

En el cuadrado de arriba a la izquierda sí que vemos 4 superficies continuas bastante bien definidas y de acuerdo a lo esperado. Es de esperar que, a mayor similitud y mayor proporción de contenedores bien recogidos, la bondad de la orden sea mayor y la orden pertenecerá a una clase superior, reservando la clase 3 (órdenes ***muy buenas***) a la esquina superior derecha del cuadrado. De igual modo, las órdenes ***malas*** cabe esperarlas en la esquina inferior izquierda o, en este caso, en todo el tercio izquierdo debido al dominio que la variable `BienHechos` tiene sobre la variable `Similitud`. No todos los puntos encajan en la superficie que deberían corresponderles, pero las ideas principales han sido capturadas.

Revisando en cuadrado central superior, se refleja de nuevo el dominio de la variable `BienHechos` por la verticalidad de las superficies. Sin embargo, vemos que la clase 3 se reserva a un pequeño trozo de la esquina inferior derecha del cuadrado. Esto quiere decir que el modelo ha capturado la idea de que, incluso si estamos premiando que la bondad se vea favorecida por un valor alto de la variable `Adicionales`, efectivamente las órdenes de clase 3 tendrán un valor bajo de la variable `Adicionales`. Esto es porque, en general, las órdenes de trabajo con un alto valor de la variable `BienHechos` debido a un cumplimiento estricto de la orden suelen desviarse poco hacia la recogida de contenedores adicionales.

En los demás casos, la información es más caótica y poco fiable. Queda claro en algunos casos que el entrenamiento con dos variables es insuficiente. Especialmente, cuando se trata de variables con menor contribución a la bondad total como es el caso del cuadrado de abajo a la derecha, la información de las superficies es mucho más imprecisa y los puntos no caen donde deben. Este método simplemente nos da una primera idea superficial de la naturaleza de las variables.

D.5. *Extreme Gradient Boosting*

Dentro de los métodos de *ensemble*, encontramos dos grupos: las técnicas de aprendizaje paralelo y las técnicas de aprendizaje secuencial. El ejemplo más claro del primer tipo son los bosques aleatorios que toman una serie de árboles independientes y se reduce el error al tener en cuenta el rendimiento de múltiples árboles. Sin embargo, también merece la pena el estudio de técnicas del segundo tipo. En el caso de trabajar con árboles, los árboles entrenados se generarían en una secuencia con cierta dependencia entre ellos para que los nuevos árboles que vayan generándose aprendan de los errores de los árboles anteriores.

Los algoritmos de *gradient boosting* pertenecen al segundo tipo y son una técnica muy popular de clasificación y regresión. En principio, los algoritmos de *extreme gradient boosting* de la librería

`XGBoost` están mejor optimizados y regularizados para la prevención del sobreajuste de modelos.

Para los propósitos de este trabajo, se ha probado la implementación del clasificador `XGBClassifier` y del regresor `XGBRegressor`. No obstante, se ha observado que, para este conjunto de datos, los resultados igualan a los del bosque aleatorio sin aportar información de mayor relevancia, por lo que no se han desarrollado en mayor profundidad en el trabajo. Quizá para conjuntos de datos más grandes y complejos cabría esperar mejoras notables en el rendimiento.

E. Acerca de las redes neuronales

Una representación esquemática de la forma de una de estas redes podría ser la siguiente:

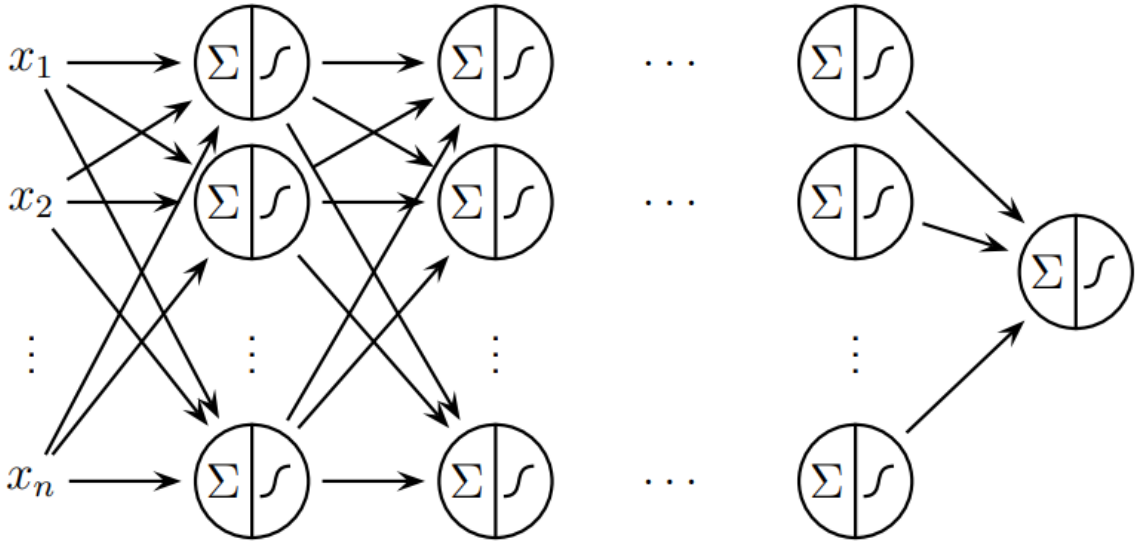

Figura 8: Perceptrón multicapa.

donde se refleja los dos pasos que se dan en cada neurona: realizar una combinación lineal y aplicar una función no lineal.

A continuación se describen algunos algoritmos fundamentales que forman parte del funcionamiento de las redes neuronales.

E.1. Descenso de gradiente

El descenso de gradiente es un algoritmo iterativo de optimización para hallar un mínimo local de una función. Esto se hace moviéndonos en la dirección de máximo decrecimiento que viene dada por el gradiente cambiado de signo en cada punto. Iniciando en un punto, calculamos el gradiente en ese punto, lo cambiamos de signo y, con un cierto paso, pasamos al siguiente punto donde repetiremos el proceso. Esto hará que, si se ha elegido un paso adecuado para la

función, finalmente nos aproximemos lo suficiente a un mínimo local de la función. El tamaño del paso depende de un parámetro conocido como *learning rate* (α) que multiplica al gradiente. Debe elegirse cuidadosamente porque un valor demasiado alto puede hacer que sobrepasemos el mínimo local sin llegar a alcanzarlo, mientras que un valor demasiado bajo puede requerir un tiempo de computación extremadamente grande.

Definimos como siempre una función de coste J que será de la que calcularemos los gradientes (vectores de derivadas con respecto a los parámetros de la red) ya que queremos minimizarla. Así, en cada iteración, el valor de un conjunto de parámetros θ en función del valor de los de la iteración anterior viene dado por

$$\theta_{i+1} = \theta_i - \alpha \nabla J(\theta_i) \tag{4}$$

En *machine learning*, este algoritmo se usa para actualizar los parámetros de un modelo. Por ejemplo, los pesos de una red neuronal.

E.2. Retropropagación

El cálculo de los gradientes descritos en el método anterior no es trivial debido al alto número de parámetros que pueden existir en una red neuronal y su distribución en las múltiples capas. Por ello, se hace uso de un método conocido como *back-propagation* o retropropagación. Calculamos las derivadas parciales de la función de coste con respecto a los parámetros de la última capa aplicando la regla de la cadena. Después, vamos aplicando el mismo método a las capas anteriores una a una hasta llegar al inicio de la red.

Para simplificar la explicación del concepto, supongamos que tenemos una red con sólo una neurona en cada capa. Tal y como se describe en el trabajo, en cada neurona se dan dos operaciones:

$$z^{(n)} = w^{(n)} a^{(n-1)} + b^{(n)} \tag{5}$$

$$a^{(n)} = \phi^{(n)} \left(z^{(n)} \right) \tag{6}$$

Supongamos que en el método del descenso de gradiente necesitamos calcular la derivada de la función de coste con respecto al peso $w^{(n)}$. Entonces, esto es una simple aplicación de la regla de la cadena:

$$\frac{\partial J_k}{\partial w^{(n)}} = \frac{\partial z^{(n)}}{\partial w^{(n)}} \cdot \frac{\partial a^{(n)}}{\partial z^{(n)}} \cdot \frac{\partial J_k}{\partial a^{(n)}} \tag{7}$$

Ahora habría que calcular cada una de esas tres derivadas. Supongamos que la función de coste asociada a una determinada muestra del conjunto de entrenamiento viene dada por la diferencia cuadrática entre el valor de la activación a en la capa n previa a la salida y el valor deseado en la neurona de salida para dicha muestra específica del conjunto de datos, y_k. Es decir,

$$J_k = \left(a^{(n)} - y_k \right)^2 \tag{8}$$

Entonces, a partir de las expresiones (5), (6) y (8), calculamos las tres derivadas que resultan ser

$$\frac{\partial z^{(n)}}{\partial w^{(n)}} = a^{(n-1)} \tag{9}$$

$$\frac{\partial a^{(n)}}{\partial z^{(n)}} = \left(\phi^{(n)}\right)' \left(z^{(n)}\right) \tag{10}$$

$$\frac{\partial J_k}{\partial a^{(n)}} = 2\left(a^{(n)} - y_k\right) \tag{11}$$

Realizado este proceso para una cierta muestra del conjunto de datos, la derivada parcial de la función de coste total sería el promedio de todas las derivadas parciales obtenidas para todas las muestras del conjunto de datos.

Equivalentemente, la derivada parcial con respecto al *bias* $b^{(n)}$ es

$$\frac{\partial J_k}{\partial b^{(n)}} = \frac{\partial z^{(n)}}{\partial b^{(n)}} \cdot \frac{\partial a^{(n)}}{\partial z^{(n)}} \cdot \frac{\partial J_k}{\partial a^{(n)}} \tag{12}$$

donde

$$\frac{\partial z^{(n)}}{\partial b^{(n)}} = 1 \tag{13}$$

De nuevo, realizado este proceso para una cierta muestra del conjunto de datos, la derivada parcial de la función de coste total sería el promedio sobre el conjunto de datos.

Para redes con más neuronas por capa, el proceso es análogo teniendo en cuenta los múltiples pesos extra que aparecen y las diferentes activaciones que aparecen en una misma capa.

E.3. L-BFGS

`L-BFGS` (*Limited-memory Broyden–Fletcher–Goldfarb–Shanno*) es un algoritmo de optimización empleado en redes neuronales. Se trata de un método de segundo orden. Pertenece a la familia de los métodos cuasi-Newton. Esto último quiere decir que se realizan aproximaciones de la inversa de la matriz hessiana en lugar de su cálculo explícito. Es una mejora del algoritmo `BFGS` ya que requiere guardar menos información en memoria en cada iteración, lo cual es relevante en términos de eficiencia. En principio, para conjuntos de datos de poco tamaño, es de esperar un mejor rendimiento a través de este resolvedor que a través de otros típicamente utilizados en el entrenamiento de redes neuronales como el descenso de gradiente estocástico o `Adam`.

E.4. Adam

Se trata de una modificación del descenso de gradiente estocástico clásico que combina las ventajas de otras populares extensiones del descenso de gradiente estocástico como `AdaGrad` o `RMSprop`. Se calculan *learning rates* adaptables para los diferentes parámetros a partir de

estimaciones del primer y segundo momento de los gradientes (asumiendo que los gradientes son variables aleatorias). Esencialmente, cuenta con 4 hiperparámetros de interés:

- α: Tamaño de paso.
- β_1: Ritmo de caída exponencial para las estimaciones del primer momento.
- β_2: Ritmo de caída exponencial para las estimaciones del segundo momento.
- ε: Tolerancia para evitar divisiones entre cero.

Se trata de uno de los métodos más eficientes y es especialmente adecuado en problemas grandes en términos de número de datos y número de parámetros.

E.5. Implementación de las redes

Para crear una red neuronal con el método `L-BFGS` mientras se compara el rendimiento para distintas combinaciones de parámetros, puede utilizarse el siguiente código:

```
X = variables[['BienHechos','Similitud','Adicionales','RatioLongitudes']]
y = variables['Resultado']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

mlp = MLPClassifier()
parameter_space = {
    'hidden_layer_sizes': [(4,),(5,),(6,)],
    'activation': ['tanh','logistic','relu'],
    'solver': ['lbfgs'],
    'alpha': [0.01,0.001,0.0001,0.00001],
    'max_iter': [1000],
    'learning_rate': ['constant'],
}
clf = GridSearchCV(mlp, parameter_space, n_jobs=-1, cv=3)
clf.fit(X_train, y_train)

print('Mejores parametros:\n', clf.best_params_)
```

donde puede hacerse uso del buscador de parámetros `GridSearchCV` que ya aplica validación cruzada (*cross validation*) mediante `K-Fold`. Este método realiza K particiones distintas del conjunto de datos total en subconjuntos de entrenamiento y de test para evaluar la precisión de acuerdo a cada una de las particiones y promedia los resultados.

Asimismo la implementación de `Adam` en `TensorFlow` se lleva a cabo a través de un adecuado tratamiento de la clase de tensores típica de la librería:

```
def get_batch(x_data, y_data, batch_size):
    idxs = np.random.randint(0, len(y_data), batch_size)
    return x_data[idxs,:], y_data[idxs]

def nn_model(x_input, W1, b1, W2, b2):
    x_input = tf.reshape(x_input, (x_input.shape[0], -1))
    x = tf.add(tf.matmul(tf.cast(x_input, tf.float32), W1), b1)
    x = tf.nn.relu(x)
    logits = tf.add(tf.matmul(x, W2), b2)
    return logits

def loss_fn(logits, labels):
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    labels=labels,

    logits=logits))
    return cross_entropy

epochs = 80
batch_size = 50

X = variables[['BienHechos','Similitud','Adicionales','RatioLongitudes']]
y = variables['Resultado']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

X_train = tf.Variable(np.array(X_train).astype('float'))
X_test = tf.Variable(np.array(X_test).astype('float'))

#Pesos entre la capa de entrada y la capa oculta
W1 = tf.Variable(tf.random.normal([4, 4], stddev=0.03), name='W1')

#Bias de la capa oculta
b1 = tf.Variable(tf.random.normal([4]), name='b1')

#Pesos entre la capa oculta y la capa de salida
W2 = tf.Variable(tf.random.normal([4, 4], stddev=0.03), name='W2')

#Bias de la capa de salida
b2 = tf.Variable(tf.random.normal([4]), name='b2')

optimizer = tf.keras.optimizers.Adam(learning_rate=0.07,beta_1=0.9,beta_2=0.999,
    epsilon=0.0000001)

total_batch = int(len(y_train) / batch_size)
arrayloss=[]
arrayacc=[]
for epoch in range(epochs):
    avg_loss = 0
    for i in range(total_batch):
        batch_x, batch_y = get_batch(np.array(X_train), np.array(y_train),
    batch_size=batch_size)

        batch_x = tf.Variable(np.array(batch_x).astype('float'))
```

```
        batch_y = tf.Variable(np.array(batch_y).astype('float'))

        batch_y = tf.one_hot(np.array(batch_y).astype('int'),4)
        with tf.GradientTape() as tape:
            logits = nn_model(batch_x, W1, b1, W2, b2)
            loss = loss_fn(logits, batch_y)
        gradients = tape.gradient(loss, [W1, b1, W2, b2])
        optimizer.apply_gradients(zip(gradients, [W1, b1, W2, b2]))
        avg_loss += loss / total_batch
    arrayloss.append(avg_loss)
    test_logits = nn_model(X_test, W1, b1, W2, b2)
    max_idxs = tf.argmax(test_logits, axis=1)
    test_acc = np.sum(max_idxs.numpy() == y_test) / len(y_test)
    arrayacc.append(test_acc)
    print(f"Epoca: {epoch + 1}, perdida={avg_loss:.3f}, precision={test_acc
    *100:.3f}%")
```

Referencias adicionales

La bibliografía principal se ha presentado en el trabajo, pero a continuación se muestran algunas referencias de interés acerca de los conceptos de este anexo.

[1] Bonde, O. y Karlsson, L. (2020). *A Comparison of Selected Optimization Methods for Neural Networks.* https://www.diva-portal.org/smash/get/diva2:1438308/FULLTEXT01.pdf

[2] Brownlee, J. (2017). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.* https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning

[3] Chen, T. y Guestrin, C. (2016). *XGBoost: A Scalable Tree Boosting System.* https://arxiv.org/pdf/1603.02754.pdf

[4] Pandey, P. (2019). *Understanding the Mathematics behind Gradient Descent.* https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e

[5] Sanderson, G. (2017). *Backpropagation calculus — Deep learning, chapter 4.* https://www.youtube.com/watch?v=tIeHLnjs5U8